آموزش جامع ساختمان داده و الگوریتم (DSA) — مفهومی و عملی با پایتون

جدول پیشرفت یادگیری DSA

مبحث	وضعیت
آرایه (List/Array) در پایتون	[ ]
لیست پیوندی	[ ]
استک (Stack)	[ ]
صف (Queue و Priority/Circular)	[ ]
دیکشنری/هش‌مپ (Dict/HashMap)	[ ]
مجموعه (Set)	[ ]
تحلیل پیچیدگی زمانی و فضایی (Big O)	[ ]
جستجو (خطی، دودویی)	[ ]
مرتب‌سازی	[ ]
دک (Deque)	[ ]
درخت‌ها (Binary tree، BST، Traversal)	[ ]
درخت‌های ویژه (Heap, AVL, Trie)	[ ]
گراف‌ها (نمایش، کاربرد، BFS/DFS)	[ ]
الگوریتم‌های گراف (کوتاه‌ترین مسیر و MST)	[ ]
بازگشت (Recursion)	[ ]
تمرین‌های ترکیبی، مینی‌پروژه ساده	[ ]
ضمیمه: توابع کلیدی پایتون، منابع آنلاین	[ ]

فهرست مطالب

3. استک (Stack)
4. صف (Queue و Priority/Circular)
5. دیکشنری/هش‌مپ (Dict/HashMap)
6. مجموعه (Set)
7. تحلیل پیچیدگی زمانی و فضایی (Big O)
8. جستجو (خطی، دودویی)
9. مرتب‌سازی (Selection, Bubble, Insertion, Quick, Merge, Heap sort)
10. دک (Deque)
11. درخت‌ها (Binary tree، BST، Traversal)
12. درخت‌های ویژه (Heap, AVL, Trie)
13. گراف‌ها (نمایش، کاربرد، BFS/DFS)
14. الگوریتم‌های گراف (کوتاه‌ترین مسیر و MST)
15. بازگشت (Recursion)
16. تمرین‌های ترکیبی و مینی‌پروژه
17. ضمیمه: توابع کلیدی پایتون، منابع توصیه شده

فصل 1: آرایه (List/Array) در پایتون

تعریف دقیق با مثال مفهومی

آرایه یا لیست مهم‌ترین و پرکاربردترین ساختمان داده خطی است که مجموعه‌ای از عناصر مشابه یا غیرهمنوع را به ترتیب مشخصی ذخیره می‌کند. در پایتون، لیست‌ها نوع پیش‌فرض آرایه‌ها هستند که می‌توانند مقادیر مختلف را در کنار هم نگهداری کنند و از نظر اندازه پویا هستند.

مثال:

یک لیست از نام دانش‌آموزان:

```
students = ["علی", "سارا", "محمد", "زهرا"]
```

دسترسی به اعضای لیست

برای دسترسی به عضوی از لیست از اندیس استفاده می‌کنیم:

```
print(students[0])  # خروجی: علی
```

اندیس‌گذاری در پایتون از صفر شروع می‌شود.

تغییر مقدار یک عضو

```
students[1] = "مینا"
print(students)  # خروجی: ['علی', 'مینا', 'محمد', 'زهرا']
```

حذف عضو

با استفاده از `del` یا `remove` :

```
del students[2]  # حذف عضو سوم (محمد)
print(students)  # خروجی: ['علی', 'مینا', 'زهرا']

students.remove("زهرا")
print(students)  # خروجی: ['علی', 'مینا']
```

پیمایش و چاپ اعضا

```
for student in students:
    print(student)
```

چند مثال دقیق‌تر

```
# اضافه کردن عضو
students.append("رضا")
# درج عضو در موقعیت مشخص
students.insert(1, "لیلا")

# با استفاده از حلقه اندیس
```

```
:for i in range(len(students))
("{students[i]} :{i} عضو شماره"f)print
```

تمرینات فصل 1

1. یک لیست از اعداد صحیح بسازید و مجموع آن‌ها را محاسبه و چاپ کنید.
2. برنامه‌ای بنویسید که از کاربر چند نام دریافت کند و در لیستی ذخیره کند سپس همه نام‌ها را با شماره چاپ کند.
3. عضو خاصی را از لیست حذف کرده و لیست جدید را نمایش دهید.

پاسخ تمرینات فصل 1

.1

```
numbers = [1, 3, 5, 7, 9]
total = sum(numbers)
(total ,"مجموع اعداد:")print
```

.2

```
[] = names
((" چند نام می‌خواهید وارد کنید؟")input)n = int
:for _ in range(n)
(" :نام")name = input
names.append(name)

:for i, name in enumerate(names)
print(f"{i+1}. {name}")
```

.3

```
["زهرا" ,"محمد" ,"سارا" ,"علی"] = names
("محمد")names.remove
print(names)  # خروجی: ['علی', 'سارا', 'زهرا']
```

فصل 2: لیست پیوندی (Linked List)

تعریف دقیق با مثال مفهومی

لیست پیوندی، ساختاری است متشکل از گره‌هایی که هر گره شامل داده و ارجاع به گره بعدی است. این ساختار امکان درج و حذف در هر موقعیت را با کارایی مناسب فراهم می‌کند. سه نوع پرکاربرد لیست پیوندی:

- یک‌طرفه (Singly Linked List): هر گره به گره بعدی اشاره می‌کند.
- دوطرفه (Doubly Linked List): هر گره به گره قبلی و بعدی اشاره دارد.
- حلقوی (Circular Linked List): انتهای لیست مجدداً به سر لیست وصل می‌شود.

مقایسه اجمالی:

نوع لیست	دسترسی	درج/حذف اول	حافظه اضافی
یک‌طرفه	خطی	سریع	کم
دوطرفه	خطی	سریع‌تر	بیشتر
حلقوی	خطی	مناسب برای ساختارهای دایره‌ای مشابه نوع اول	

پیاده‌سازی ساده لیست پیوندی یک‌طرفه با پایتون

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
```

```
                return
        current = self.head
        :while current.next
    current = current.next
    current.next = new_node

        :def display(self)
        current = self.head
        :while current
print(current.data, end=" -> ")
    current = current.next
        print("None")
```

استفاده:

```
()ll = SinglyLinkedList
ll.append(10)
ll.append(20)
ll.append(30)
()ll.display
# خروجی: 10 -> 20 -> 30 -> None
```

پیاده‌سازی لیست پیوندی دوطرفه (Doubly Linked List)

```
:class Node
:def __init__(self, data)
    self.data = data
    self.next = None
    self.prev = None

:class DoublyLinkedList
:def __init__(self)
    self.head = None
```

```
                    :def append(self, data)
                  new_node = Node(data)
                      :if not self.head
                self.head = new_node
                             return
                      current = self.head
                      :while current.next
                current = current.next
                   current.next = new_node
                   new_node.prev = current

                    :def display_forward(self)
                       current = self.head
                            :while current
        print(current.data, end=" <-> ")
                  current = current.next
                              print("None")

                   :def display_backward(self)
                       current = self.head
                          :if not current
                               return
                      :while current.next
                current = current.next
                            :while current
        print(current.data, end=" <-> ")
                  current = current.prev
                              print("None")
```

لیست پیوندی حلقوی (Circular Linked List) یک‌طرفه

```
                                :class Node
                    :def __init__(self, data)
                         self.data = data
                         self.next = None
```

```
:class CircularLinkedList
  :def __init__(self)
   self.head = None

 :def append(self, data)
  new_node = Node(data)
    :if not self.head
   self.head = new_node
  self.head.next = self.head
              return
    current = self.head
:while current.next != self.head
   current = current.next
     current.next = new_node
   new_node.next = self.head

      :def display(self)
    :if not self.head
  ("لیست خالی است.")print
              return
    current = self.head
           :while True
print(current.data, end=" -> ")
   current = current.next
 :if current == self.head
            break
  ("(برگشت به ابتدا)")print
```

تفاوت و کاربردها

نوع لیست	مزایا	معایب	کاربردها
یک‌طرفه	ساده، حافظه کمتر	دسترسی فقط به جلو	پیاده‌سازی صف و استک ساده
دوطرفه			مرورگرها، undo/redo

نوع لیست	مزایا	معایب	کاربردها
	پیمایش دوطرفه، حذف ساده‌تر	پیچیدگی بیشتر، حافظه بیشتر	
حلقوی	پیمایش دایره‌ای، مناسب حلقه	پیچیده‌تر در مدیریت	مدیریت منابع چرخشی، بازی‌ها

تمرینات فصل 2

1. لیست پیوندی یک‌طرفه بسازید و تابع حذف یک عضو خاص را پیاده کنید.
2. لیست دوطرفه بسازید و لیست را از انتها به ابتدا چاپ کنید.
3. لیست حلقوی بسازید و چاپ دایره‌ای اعضا را پیاده کنید.

پاسخ تمرینات فصل 2

.1

```
                                :class Node
                    :def __init__(self, data)
                         self.data = data
                         self.next = None

                          :class SinglyLinkedList
                          :def __init__(self)
                         self.head = None

                       :def append(self, data)
                    new_node = Node(data)
                        :if not self.head
                self.head = new_node
                             return
                      current = self.head
                      :while current.next
            current = current.next
               current.next = new_node
```

```
                        :def remove(self, key)
                    current = self.head
                            prev = None
    :while current and current.data != key
                        prev = current
                current = current.next
                        :if not current
                # عنصر پیدا نشد  return
                            :if not prev
# حذف سر لیست  self.head = current.next
                                    :else
                prev.next = current.next

                            :def display(self)
                    current = self.head
                            :while current
        print(current.data, end=' -> ')
                current = current.next
                            print('None')

                        ()ll = SinglyLinkedList
                                ll.append(5)
                               ll.append(10)
                               ll.append(15)
                                ()ll.display
                               ll.remove(10)
                                ()ll.display
```

.2

```
 # استفاده از کلاس DoublyLinkedList در بالا:
                        ()dll = DoublyLinkedList
                                dll.append(1)
                                dll.append(2)
                                dll.append(3)
                    print("چاپ از ابتدا به انتها:")
                                ()dll.display_forward
```

```
print("چاپ از انتها به ابتدا:")
dll.display_backward()
```

.3

```
cll = CircularLinkedList()
cll.append("a")
cll.append("b")
cll.append("c")
cll.display()
```

فصل 3: استک (Stack)

تعریف دقیق با مثال مفهومی

استک یک ساختمان داده LIFO (Last In First Out) است. داده‌ها به صورت پشته روی هم قرار می‌گیرند و فقط دسترسی به آخرین ورودی وجود دارد.

مثال در دنیای واقعی: عملیات Undo در نرم‌افزارها.

پیاده‌سازی استک به صورت ساده در پایتون با لیست

```
stack = []

# افزودن به استک (Push)
stack.append(5)
stack.append(10)

# برداشتن از استک (Pop)
item = stack.pop()
print(item)  # خروجی: 10

# مشاهده عنصر بالا بدون حذف
```

```
top = stack[-1]
print(top)  # خروجی: 5
```

پیاده‌سازی استک با کلاس

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            return None

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            return None

    def is_empty(self):
        return len(self.items) == 0

    def size(self):
        return len(self.items)
```

مثال کاربردی: undo ساده

```
stack = Stack()
stack.push("حالت اول")
```

```
stack.push("حالت دوم")
stack.push("حالت سوم")

print("وضعیت فعلی:", stack.peek())

stack.pop()
print("بعد از undo:", stack.peek())
```

تمرینات فصل 3

1. برنامه‌ای بنویسید که پرانتزهای یک عبارت ریاضی را بررسی کند (باز و بسته بودن درست).
2. استک را با نهایی کردن حداقل المنت در هر لحظه گسترش دهید (ساختار MinStack).
3. آکادمی چاپ معکوس کلمه با استفاده از استک را پیاده کنید.

پاسخ تمرینات فصل 3

.1

```
def is_balanced(expression):
    stack = []
    pairs = {'(': ')', '[': ']', '{': '}'}
    for char in expression:
        if char in "([{":
            stack.append(char)
        elif char in "}])":
            if not stack or stack[-1] != pairs[char]:
                return False
            stack.pop()
    return len(stack) == 0

expr = "(a+b)*[c/d]"
print(is_balanced(expr))  # خروجی: True
```

.2

```
                                        :class MinStack
                                    :def __init__(self)
                                    [] = self.stack
                                [] = self.min_stack

                                    :def push(self, val)
                                self.stack.append(val)
:if not self.min_stack or val <= self.min_stack[-1]
                        self.min_stack.append(val)

                                        :def pop(self)
                                ()val = self.stack.pop
                        :if val == self.min_stack[-1]
                             ()self.min_stack.pop
                                        return val

                                    :def get_min(self)
                                :if self.min_stack
                        return self.min_stack[-1]
                                    return None

                                        ()ms = MinStack
                                             ms.push(3)
                                             ms.push(5)
                                             ms.push(2)
                                print(ms.get_min())  # خروجی: 2
                                               ()ms.pop
                                print(ms.get_min())  # خروجی: 3
```

.3

```
                                    :def reverse_word(word)
                                              [] = stack
                                         :for ch in word
                                    stack.append(ch)
                                      '' = reversed_word
                                           :while stack
```

```
                    ()reversed_word += stack.pop
                        return reversed_word

print(reverse_word("python"))  # nohtyp
```

فصل 4: صف (Queue و Priority/Circular Queue)

تعریف دقیق با مثال مفهومی

صف یک ساختمان داده FIFO (First In First Out) است. داده‌ها به ترتیب وارد شده پردازش می‌شوند.

نمونه واقعی: صف نوبت‌دهی بانک.

صف ساده با لیست پایتون

```
from collections import deque

()queue = deque

("مشتری 1")queue.append
("مشتری 2")queue.append

print(queue.popleft())  # خروجی: مشتری 1
print(queue.popleft())  # خروجی: مشتری 2
```

صف با کلاس خودساخته

```
class Queue:
    def __init__(self):
```

```
                                    [] = self.items

                                :def enqueue(self, item)
                            self.items.append(item)

                                     :def dequeue(self)
                                     :if self.items
                        return self.items.pop(0)
                                        return None

                                     :def is_empty(self)
                         return len(self.items) == 0
```

صف حلقوی (Circular Queue)

در صف حلقوی بعد از رسیدن به انتها، مجددا از ابتدای صف استفاده می‌شود تا حافظه بهینه گردد.

صف اولویت‌دار (Priority Queue)

در صف اولویت‌دار عناصر بر اساس اولویت خاص پردازش می‌شوند نه فقط ترتیب ورود.

پیاده‌سازی ساده با heapq

```
                                          import heapq

                                                [] = pq

                    ((وظیفه کم اولویت" ,2) ,pq)heapq.heappush
                ((وظیفه با اولویت بالا" ,1) ,pq)heapq.heappush

                                             :while pq
                        priority, task = heapq.heappop(pq)
                                             print(task)
```

تمرینات فصل 4

1. صف ساده خودساخته پیاده کنید و چند عنصر اضافه و حذف کنید.
2. برنامه‌ای بنویسید که صفی از نام‌ها بگیرد و به ترتیب پردازش کند.
3. صف اولویت‌دار برای تضمین اجرای تسک‌ها بر اساس اولویت ساخته و تست کنید.

پاسخ تمرینات فصل 4

.1

```
q = Queue()
q.enqueue("الف")
q.enqueue("ب")
print(q.dequeue())  # خروجی: الف
print(q.dequeue())  # خروجی: ب
```

.2

```
from collections import deque
names_queue = deque()
for _ in range(3):
    name = input("نام فرد بعدی در صف: ")
    names_queue.append(name)

while names_queue:
    print(f"خدمت به: {names_queue.popleft()}")
```

.3

```
import heapq
tasks = []
heapq.heappush(tasks, (3, "مطالعه"))
heapq.heappush(tasks, (1, "پروژه برنامه‌نویسی"))
heapq.heappush(tasks, (2, "تمرین ریاضی"))
```

```
:while tasks
        priority, task = heapq.heappop(tasks)
("{priority} اولویت با {task} اجرای وظیفه: "f)print
```

فصل 5: دیکشنری/هش‌مپ (Dict/HashMap)

تعریف دقیق با مثال مفهومی

دیکشنری یا هش مپ ساختاری است که به صورت کلید-مقدار (key-value) داده‌ها را ذخیره می‌کند. هر کلید به یک مقدار نگاشت می‌شود و دسترسی به مقادیر بر اساس کلید سریع است.

نحوه کار هش مپ

- داده‌ها براساس کلیدهای هش‌شده ذخیره می‌شوند.
- برخورد (Collision) در صورت هش یکسان توسط روش‌های مختلف حل می‌شود (زنجیره‌ای یا آدرس‌دهی باز).

پیاده‌سازی نمونه دیکشنری در پایتون

```
} = student_grades
    "علی": 18,
    "سارا": 20,
    "محمد": 15
{

print(student_grades["سارا"])  # خروجی: 20

student_grades["زهرا"] = 19  # افزودن یک دانش‌آموز جدید
```

مزایا و معایب دیکشنری

مزایا	معایب
دسترسی سریع به مقادیر	استفاده بیشتر از حافظه
ذخیره‌سازی کلید-مقدار منعطف	ترتیب عناصر در نسخه‌های قدیمی پایتون مشخص نیست
افزودن/حذف آسان عناصر	کلیدها باید غیرقابل تغییر (immutable) باشند

نمونه کد هش‌مپ ساده (با زنجیره‌ای کردن)

```
class HashMap:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        return hash(key) % self.size

    def set(self, key, value):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)
                return
        self.table[index].append((key, value))

    def get(self, key):
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v
        return None

h = HashMap()
h.set("علی", 21)
h.set("سارا", 22)
```

```
print(h.get("علی"))  # خروجی: 21
```

تمرینات فصل 5

1. یک دیکشنری از کشورها و پایتخت‌های آن‌ها بسازید و یکی را حذف کنید.
2. برنامه‌ای بسازید که تعداد تکرار هر کلمه در متن را بشمارد.
3. هش‌مپ ساده با زنجیره‌ای کردن پیاده‌سازی کنید و عملیات درج و جستجو انجام دهید.

پاسخ تمرینات فصل 5

.1

```
capitals = {"ایران": "تهران", "فرانسه": "پاریس", "ژاپن": "توکیو"}
del capitals["فرانسه"]
print(capitals)  # خروجی: {'ایران': 'تهران', 'ژاپن': 'توکیو'}
```

.2

```
text = "این یک متن ساده است و این یک مثال است"
words = text.split()
freq = {}
for w in words:
    freq[w] = freq.get(w, 0) + 1
print(freq)
```

1. (مشابه کد بالا در hash map ساده)

فصل 6: مجموعه (Set)

تعریف دقیق با مثال مفهومی

مجموعه ساختاری است که مقادیر یکتا و بدون ترتیب خاص را نگهداری می‌کند و برای عملیات مانند اجتماع، اشتراک، تفاضل بسیار کاربردی است.

تعریف و استفاده

```
s = set()
s.add(1)
s.add(2)
s.add(2)  # اضافه شدن مجدد تکراری تاثیری ندارد

print(s)  # خروجی: {1, 2}
```

عملیات متداول

```
a = {1, 2, 3}
b = {2, 3, 4}

print(a.union(b))      # {1, 2, 3, 4}
print(a.intersection(b))  # {2, 3}
print(a.difference(b))  # {1}
```

تمرینات فصل 6

1. از دو لیست، مجموعه اعداد مشترک را بیابید.
2. با استفاده از مجموعه، لیستی بدون تکرار بسازید.
3. برنامه‌ای بنویسید که بررسی کند آیا یک مجموعه زیرمجموعه مجموعه دیگر است یا خیر.

پاسخ تمرینات فصل 6

.1

```
list1 = [1,2,3,4]
list2 = [3,4,5,6]
s1 = set(list1)
s2 = set(list2)
print(s1.intersection(s2))  # خروجی: {3,4}
```

.2

```
lst = [1, 2, 2, 3, 3, 3]
unique_list = list(set(lst))
print(unique_list)
```

.3

```
a = {1, 2}
b = {1, 2, 3, 4}
print(a.issubset(b))  # خروجی: True
```

فصل 7: تحلیل پیچیدگی زمانی و فضایی (Big O)

تعریف دقیق

Big O روشی برای توصیف میزان افزایش منابع مصرفی (زمان یا حافظه) الگوریتم بر اساس اندازه ورودی است.

تعاریف مهم

نماد Big O	توضیح	مثال
O(1)	زمان ثابت، مستقل از اندازه ورودی	دسترسی به اندیس آرایه
O(log n)	زمان لگاریتمی، مثل جستجوی دودویی	جستجوی دودویی در آرایه مرتب
O(n)	زمان خطی، متناسب با اندازه ورودی	پیمایش آرایه
O(n log n)	زمان تقریبی مرتب‌سازی‌های کاراتر	مرتب سازی سریع (Quicksort)
O(n²)	زمان درجه دو، اغلب در حلقه‌های تو در تو	الگوریتم انتخاب ساده

جدول مقایسه نمونه الگوریتم‌ها

الگوریتم	حداقل پیچیدگی	حداکثر پیچیدگی	پیچیدگی فضایی
جستجوی خطی	O(1)	O(n)	O(1)
جستجوی دودویی	O(1)	O(log n)	O(1)
مرتب‌سازی حباب	O(n)	O(n²)	O(1)
مرتب‌سازی سریع	O(n log n)	O(n²)	O(log n)
مرتب‌سازی ادغامی	O(n log n)	O(n log n)	O(n)

مثال ملموس: جستجوی عدد در لیست

- جستجوی خطی: هر عدد را از ابتدا تا انتها چک می‌کند. زمان در بدترین حالت O(n).
- جستجوی دودویی: فقط اگر لیست مرتب باشد امکان دارد. قسمت میانی انتخاب و رد نیمه‌ها انجام می‌شود. زمان O(log n).

تمرینات فصل 7

1. برای الگوریتم درج در آرایه چه پیچیدگی زمانی وجود دارد؟
2. مرتب‌سازی حباب را بنویسید و پیچیدگی آن را تحلیل کنید.
3. زمان اجرای جستجوی دودویی در لیست مرتب طول ۱۶ را گام به گام توضیح دهید.

پاسخ تمرینات فصل 7

1. درج در انتهای آرایه: O(1) متوسط، درج در وسط آرایه: O(n)
2.

```
def bubble_sort(lst):
    n = len(lst)
    for i in range(n):
        for j in range(0, n-i-1):
            if lst[j] > lst[j+1]:
                lst[j], lst[j+1] = lst[j+1], lst[j]
```

پیچیدگی زمانی: $O(n^2)$
3. با شروع از عدد وسط لیست (اندیس 8)، هر بار نصف نیمه بررسی می‌شود:
گام 1: مقایسه با عنصر 8
گام 2: بررسی نیمه اول یا دوم
گام 3: بررسی نیمه کوچکتر (4، 2، 1)
کل گام‌ها: حدود $\log_2(16) = 4$

فصل 8: جستجو (خطی، دودویی)

جستجوی خطی

ایده ساده: جستجوی عنصر با بررسی ترتیب در آرایه یا لیست

```
def linear_search(lst, target):
    for i, val in enumerate(lst):
        if val == target:
            return i
    return -1
```

پیچیدگی زمانی: O(n)

جستجوی دودویی

آرایه باید مرتب باشد. تکرار نصف کردن محدوده جستجو تا یافتن عنصر.

```
:def binary_search(lst, target)
  low, high = 0, len(lst)-1
         :while low <= high
 mid = (low + high) // 2
  :if lst[mid] == target
         return mid
 :elif lst[mid] < target
        low = mid+1
                  :else
       high = mid-1
                  return -1
```

پیچیدگی زمانی: O(log n)

تمرینات فصل 8

1. جستجوی خطی و دودویی را روی یک لیست نمونه امتحان کنید.
2. برنامه‌ای بنویسید که تعداد دفعات تکرار یک عدد در آرایه را بیابد.
3. اگر آرایه مرتب نباشد، جستجوی دودویی صحیح عمل می‌کند؟ چرا؟

پاسخ تمرینات فصل 8

.1

```
                  lst = [1, 3, 5, 7, 9]
print(linear_search(lst, 5))  # خروجی: 2
print(binary_search(lst, 5))  # خروجی: 2
```

.2

```
:def count_occurrences(lst, target)
                            count = 0
                        :for val in lst
                    :if val == target
                       count += 1
                             return count
```

1. خیر، جستجوی دودویی فقط روی آرایه مرتب کار می‌کند چون فرض تقسیم‌بندی درست بر اساس مرتب بودن است.

فصل 9: مرتب‌سازی (Selection, Bubble, Insertion, Quick, Merge, Heap sort)

مرتب‌سازی انتخابی (Selection Sort)

هر بار کوچک‌ترین عنصر از قسمت ناپیدا را انتخاب و به ابتدای لیست منتقل می‌کند.

```
                                :def selection_sort(lst)
                                        n = len(lst)
                                   :for i in range(n)
                                      min_idx = i
                             :for j in range(i+1, n)
                        :if lst[j] < lst[min_idx]
                                 min_idx = j
    lst[i], lst[min_idx] = lst[min_idx], lst[i]
```

پیچیدگی: $O(n^2)$

مرتب‌سازی حبابی (Bubble Sort)

مقایسه جفت به جفت و تعویض اگر نزولی است، تکرار تا مرتب شدن

```
def bubble_sort(lst):
    n = len(lst)
    for i in range(n):
        for j in range(0, n-i-1):
            if lst[j] > lst[j+1]:
                lst[j], lst[j+1] = lst[j+1], lst[j]
```

پیچیدگی: $O(n^2)$

مرتب‌سازی درج (Insertion Sort)

هر عنصر در جای درست خود در بخش مرتب شده قرار می‌گیرد.

```
def insertion_sort(lst):
    for i in range(1, len(lst)):
        key = lst[i]
        j = i-1
        while j >=0 and key < lst[j]:
            lst[j+1] = lst[j]
            j -= 1
        lst[j+1] = key
```

پیچیدگی: $O(n^2)$، اما در بهترین حالت $O(n)$

مرتب‌سازی سریع (Quick Sort)

اعمال الگوریتم تقسیم و غلبه، انتخاب یک محور (pivot)، تقسیم آرایه و مرتب‌سازی بازگشتی

```
def quick_sort(lst):
    if len(lst) <= 1:
        return lst
    pivot = lst[len(lst)//2]
    left = [x for x in lst if x < pivot]
    middle = [x for x in lst if x == pivot]
```

```
    right = [x for x in lst if x > pivot]
return quick_sort(left) + middle + quick_sort(right)
```

پیچیدگی: بهترین و متوسط O(n log n)؛ بدترین O(n²)

مرتب‌سازی ادغامی (Merge Sort)

تقسیم آرایه به دو نیمه، مرتب‌سازی هر نیمه به صورت بازگشتی و ادغام

```
:def merge_sort(lst)
:if len(lst) <= 1
    return lst
mid = len(lst)//2
left = merge_sort(lst[:mid])
right = merge_sort(lst[mid:])

    return merge(left, right)

:def merge(left, right)
    [] = result
      i = j = 0
:while i < len(left) and j < len(right)
    :if left[i] < right[j]
result.append(left[i])
        i += 1
        :else
result.append(right[j])
        j += 1
  result.extend(left[i:])
 result.extend(right[j:])
    return result
```

پیچیدگی: همیشه O(n log n)

مرتب‌سازی هیپ (Heap Sort)

ساختن یک درخت هیپ (max-heap) و استخراج MAX تکراری

```
:def heapify(lst, n, i)
        largest = i
       l = 2 * i +1
       r = 2 * i +2

:if l < n and lst[l] > lst[largest]
            largest = l
:if r < n and lst[r] > lst[largest]
            largest = r
           :if largest != i
lst[i], lst[largest] = lst[largest], lst[i]
          heapify(lst, n, largest)

           :def heap_sort(lst)
             n = len(lst)

        :for i in range(n//2 -1, -1, -1)
            heapify(lst, n, i)

          :for i in range(n-1, 0, -1)
lst[0], lst[i] = lst[i], lst[0]
            heapify(lst, i, 0)
```

پیچیدگی: O(n log n)

تمرینات فصل 9

1. پیاده‌سازی مرتب‌سازی درج و آزمایش روی یک آرایه.
2. تفاوت زمانی مرتب‌سازی بادکنکی و سریع برای آرایه‌ای با ۱۰۰۰ عنصر را بررسی کنید.
3. برنامه‌ای بنویسید که نتایج هر الگوریتم را مقایسه کند.

پاسخ تمرینات فصل 9

1. (کد درج بالا)
2. با استفاده از `time` در پایتون مقایسه و نتایج مشاهده شود (اجرای کد خارج از این مستند توصیه می‌شود).
3.

```
import time

arr = [i for i in range(1000, 0, -1)]

start = time.time()
bubble_sort(arr.copy())
print("Bubble sort:", time.time() - start)

start = time.time()
sorted_arr = quick_sort(arr.copy())
print("Quick sort:", time.time() - start)
```

فصل 10: دک (Deque)

تعریف

دک یا Double-ended queue ساختاری است که امکان افزودن و حذف در هر دو سر را دارد.

پشتیبانی پایتون از deque

```
from collections import deque
d = deque()

d.append(1)      # افزودن به انتها
d.appendleft(2)  # افزودن به ابتدا
```

```
print(d.pop())    # حذف و بازگرداندن از انتها
print(d.popleft())  # حذف و بازگرداندن از ابتدا
```

کاربردها

- پیاده‌سازی صف.
- الگوریتم‌هایی با نیاز به دسترسی دو طرف.

تمرینات فصل 10

1. صفی با امکان اضافه و حذف در دوطرف با دک بسازید.
2. برنامه‌ای با استفاده از دک ورودی کاربر را به صورت معکوس چاپ کند.

پاسخ تمرینات فصل 10

.1

```
d = deque()
d.append(10)
d.appendleft(20)
print(d)  # خروجی: deque([20, 10])
print(d.pop())  # خروجی: 10
print(d.popleft())  # خروجی: 20
```

.2

```
s = "سلام"
d = deque()
for ch in s:
    d.appendleft(ch)
reversed_s = ''.join(d)
print(reversed_s)  # مالس
```

فصل 11: درخت‌ها (Binary Tree، BST، Traversal)

تعریف دقیق

درخت ساختاری غیرخطی متشکل از گره‌ها است که هر گره می‌تواند چند فرزند داشته باشد ولی هر گره فقط یک والد دارد. درخت دودویی هر گره حداکثر دو فرزند (چپ و راست) دارد.

نمایندگی در پایتون

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

جستجوی دودویی در درخت (BST)

در BST عناصر در شاخه چپ کوچک‌تر و در شاخه راست بزرگ‌تر از گره هستند.

پیمایش‌های درخت دودویی

- پیش‌سفارش (Preorder): ریشه → چپ → راست
- میانه (Inorder): چپ → ریشه → راست
- پس‌سفارش (Postorder): چپ → راست → ریشه

نمونه کد پیمایش

```
def preorder(node):
    if node:
        print(node.data, end=' ')
        preorder(node.left)
        preorder(node.right)

def inorder(node):
    if node:
        inorder(node.left)
        print(node.data, end=' ')
        inorder(node.right)

def postorder(node):
    if node:
        postorder(node.left)
        postorder(node.right)
        print(node.data, end=' ')
```

مثال ساخت BST و پیمایش

```
root = Node(10)
root.left = Node(5)
root.right = Node(15)

preorder(root)   # خروجی: 10 5 15
inorder(root)    # خروجی: 5 10 15
postorder(root)  # 5 15 10
```

تمرینات فصل 11

1. درخت دودویی بسازید و سه نوع پیمایش را اجرا کنید.
2. تابعی برای درج عنصر جدید در BST بنویسید.

3. برنامه‌ای بنویسید که مقدار کمینه هر زیر درخت را پیدا کند.

پاسخ تمرینات فصل 11

1. (کد پیمایش بالا)
2.

```
:def insert(root, key)
   :if root is None
return Node(key)
 :if key < root.data
  root.left = insert(root.left, key)
                                :else
 root.right = insert(root.right, key)
                             return root
```

3.

```
   :def find_min(root)
     current = root
  :while current.left
current = current.left
       return current.data
```

فصل 12: درخت‌های ویژه (Heap, AVL, Trie)

Heap

ساختار درختی کامل که هر گره بزرگتر یا کوچکتر از فرزندان خود است (max-heap یا min-heap).

مثال با heapq (min-heap پیش‌فرض پایتون)

```
import heapq
[] = heap
heapq.heappush(heap, 5)
heapq.heappush(heap, 2)
print(heapq.heappop(heap))  # خروجی: 2
```

AVL Tree

درخت جستجوی دودویی خودمتعادل که اختلاف ارتفاع زیر درخت چپ و راست در هر گره حداکثر ۱ است تا پیچیدگی جستجو به O(log n) حفظ شود.

Trie

ساختار درختی برای ذخیره کلمات و جستجوی سریع بر اساس پیشوند.

پیاده‌سازی ساده

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        current = self.root
        for ch in word:
            if ch not in current.children:
                current.children[ch] = TrieNode()
            current = current.children[ch]
```

```
                current.end_of_word = True

                    :def search(self, word)
                    current = self.root
                        :for ch in word
            :if ch not in current.children
                        return False
            current = current.children[ch]
                    return current.end_of_word

                                ()trie = Trie
                            ("سلام")trie.insert
  True :خروجی #  (("سلام")trie.search)print
 False :خروجی # (("سلامت")trie.search)print
```

تمرینات فصل 12

1. کد AVL tree پایه خود را بنویسید (درج + تعادل).
2. برنامه‌ای با Trie برای ذخیره نام‌ها بسازید.
3. هیپ را به صورت max-heap پیاده کنید.

پاسخ تمرینات فصل 12

1. خارج از محدوده مستند به دلیل پیچیدگی زیاد
2. (کد trie بالا)
3. پیاده‌سازی max-heap از heapq:

```
                        import heapq
                        [] = max_heap
                heapq.heappush(max_heap, -5)
                heapq.heappush(max_heap, -1)
5 :خروجی #  print(-heapq.heappop(max_heap))
```

فصل 13: گراف‌ها (نمایش، کاربرد، BFS/DFS)

تعریف و کاربرد

گراف مجموعه‌ای از گره‌ها (راس‌ها) و یال‌ها (صلات) بین آن‌ها است. می‌تواند جهت‌دار یا بدون جهت باشد.

نمایش گراف در پایتون

- ماتریس مجاورت
- لیست مجاورتی

نمونه: دستگاه پایتون برای گراف به صورت لیست مجاورتی

```
                } = graph
    ,A': ['B', 'C']'
         ,B': ['C']'
         ,C': ['A']'
          D': ['C']'
                    {
```

پیمایش BFS (جستجوی عرضی)

استفاده از صف برای بازدید گره‌ها به ترتیب سطح.

```
from collections import deque

    :def bfs(graph, start)
       ()visited = set
queue = deque([start])
```

```
                                        :while queue
                                  ()vertex = queue.popleft
                                 :if vertex not in visited
                                print(vertex, end=" ")
                                   visited.add(vertex)
ueue.extend([n for n in graph[vertex] if n not in visited])
```

پیمایش DFS (جستجوی عمقی)

استفاده از پشته یا بازگشت برای کاوش عمیق گراف.

```
                                :def dfs(graph, start, visited=None)
                                            :if visited is None
                                            ()visited = set
                                             visited.add(start)
                                          print(start, end=" ")
                                    :for neighbor in graph[start]
                                      :if neighbor not in visited
                                dfs(graph, neighbor, visited)
```

تمرینات فصل 13

1. یک گراف نمونه با ۴ راس بسازید و BFS و DFS انجام دهید.
2. چگونگی جلوگیری از حلقه در DFS را توضیح دهید.
3. کد نمایش گراف با ماتریس مجاورت بنویسید.

پاسخ تمرینات فصل 13

.1

```
                                            } = graph
                                  ,A': ['B', 'C']'
                                       ,B': ['D']'
                                          ,[] :'C'
```

```
                                        [] :'D'
                                              {

                                bfs(graph, 'A')  # A B C D
                                                  ()print
                                dfs(graph, 'A')  # A B D C
```

1. با استفاده از مجموعه visited برای نگهداری گره‌های بازدید شده از دور زدن گره‌های قبل جلوگیری می‌شود.
3.

```
                                    nodes = ['A', 'B', 'C', 'D']
                index = {node: i for i, node in enumerate(nodes)}
                           adj_matrix = [[0]*4 for _ in range(4)]

                        edges = [('A','B'), ('B','C'), ('C','A')]

                                          :for (src, dst) in edges
                                  i, j = index[src], index[dst]
                                           adj_matrix[i][j] = 1

                                            :for row in adj_matrix
                                                   print(row)
```

فصل 14: الگوریتم‌های گراف (کوتاه‌ترین مسیر و MST)

کوتاه‌ترین مسیر (Dijkstra)

الگوریتم برای پیدا کردن کوتاه‌ترین مسیر از یک منبع به همه گره‌ها

کد ساده Dijkstra

```
import heapq

def dijkstra(graph, start):
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0
    pq = [(0, start)]

    while pq:
        current_distance, current_vertex = heapq.heappop(pq)
        if current_distance > distances[current_vertex]:
            continue
        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))
    return distances

graph = {
    'A': {'B': 5, 'C': 1},
    'B': {'A':5, 'C':2, 'D':1},
    'C': {'A':1, 'B':2, 'D':4, 'E':8},
    'D': {'B':1, 'C':4, 'E':3, 'F':6},
    'E': {'C':8, 'D':3},
    'F': {'D':6}
}
print(dijkstra(graph, 'A'))
```

درخت پوشای کمینه (MST) — الگوریتم‌های Prim و Kruskal

- Prim: شروع از راس انتخابی، افزودن کم هزینه‌ترین یال متصل به درخت

- Kruskal: مرتب سازی یال‌ها بر اساس وزن، انتخاب یال‌های کم هزینه بدون ایجاد حلقه

تمرینات فصل 14

1. الگوریتم دایجسترا روی گراف خود اجرا کنید و مسافت کوتاه را بیابید.
2. نحوه استفاده از الگوریتم کروسکال برای گراف داده شده را توضیح دهید.
3. برنامه‌ای برای MST با استفاده از الگوریتم Prim بنویسید.

پاسخ تمرینات فصل 14

1. (کد بالا کافی و آماده اجرا است)
2. ابتدا یال‌ها را بر اساس وزن مرتب کنید، سپس به ترتیب وزن یال‌ها را به مجموعه اضافه کنید اگر اضافه کردن یال حلقه ایجاد نکند.
3. پیاده‌سازی Prim خارج از این مستند به دلیل حجم است، اما مانند دایجسترا با تابع اولویت انجام می‌شود.

فصل 15: بازگشت (Recursion)

تعریف دقیق

فرایندی که در آن تابع خود را فراخوانی می‌کند تا مسئله به مسائل کوچکتر شکسته شود.

مثال سریع: محاسبه فاکتوریل

```
:def factorial(n)
    :if n == 0
      return 1
return n * factorial(n-1)

print(factorial(5))  # 120
```

قوانین بازگشت موفق

- شرط پایان
- کاهش مسئله در هر فراخوانی

تمرینات فصل 15

1. تابع فیبوناچی بازگشتی بنویسید.
2. برنامه‌ای برای معکوس کردن رشته با بازگشت بنویسید.
3. توضیح دهید چرا بازگشت بی‌پایان باعث خطا می‌شود.

پاسخ تمرینات فصل 15

.1

```
:def fibonacci(n)
   :if n <= 1
     return n
return fibonacci(n-1) + fibonacci(n-2)
```

.2

```
:def reverse_string(s)
   :if len(s) == 0
     return s
return reverse_string(s[1:]) + s[0]
```

1. بازگشت بی‌پایان باعث پر شدن پشته حافظه (Stack Overflow) می‌شود و اجرای برنامه متوقف خواهد شد.

فصل 16: تمرین‌های ترکیبی و مینی پروژه ساده

تمرین ترکیبی 1: بررسی تعادل پرانتزها با استک و بازگشت

تمرین ترکیبی 2: ساخت و جستجوی واژه‌ها با Trie و Dictionary

مینی‌پروژه: سیستم مدیریت صف بانک با Priority Queue و ثبت تراکنش‌ها

ضمیمه: توابع کلیدی پایتون برای ساختمان داده‌ها، سایت‌ها و منابع آنلاین

توابع کلیدی پایتون

- `len()`
- `append()` , `insert()` , `remove()` برای لیست
- `pop()` برای لیست و استک
- `collections.deque` برای صف و دک
- `heapq` برای صف اولویت
- `dict()` و متدهای `get()` , `keys()` , `values()`
- `set()` و عملیات اجتماع، اشتراک و تفاضل

منابع توصیه‌شده

- سایت‌های فارسی:

- وب‌سایت مکتب‌خونه
- فرادرس
- سایت‌های انگلیسی:
- GeeksforGeeks
- LeetCode
- HackerRank
- یوتیوب کانال‌های آموزشی:
- freeCodeCamp.org
- CS Dojo
- کتاب‌ها و PDFها:
- Introduction to Algorithms” by Cormen et al. (CLRS)“
- Data Structures and Algorithms Made Easy” by Narasimha Karumanchi“

راهنمای تمرین و مسیر یادگیری ادامه

- تمرین‌های مرتب و روزانه انجام دهید.
- موقع حل مسائل الگوریتمی، ابتدا تحلیل پیچیدگی زمان و فضا داشته باشید.
- پروژه‌های ساده بسازید که ساختمان داده‌ها را تمرین کنید.
- در سایت‌های حل مسئله با چالش‌های مختلف شرکت کنید.
- پس از یادگیری مفاهیم و پیاده‌سازی دستی، از کتابخانه‌ها برای بهبود عملکرد استفاده کنید.

پایان آموزش جامع ساختمان داده و الگوریتم (DSA) — مفهومی و عملی با پایتون
با آرزوی موفقیت در مسیر بزرگ مهندسی کامپیوتر و علوم داده!